TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky a mezioborových inženýrských studií

Studijní program: B2612 – Elektrotechnika a informatika
Studijní obor: 2612R011 – Elektronické informační a řídicí systémy

WYSIWYG editor pro sazbu Docbook

WYSIWYG editor for Docbook composition

Bakalářská práce

Autor: **Jindřich Růžička**
Vedoucí práce: Ing. Pavel Tyl
Konzultant: Ing. Jiří Týř

V Liberci 15. 5. 2008

Prohlášení

Byl jsem seznámen s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé bakalářské práce a prohlašuji, že **s o u h l a s í m** s případným užitím mé bakalářské práce (prodej, zapůjčení apod.).

Jsem si vědom toho, že užít své bakalářské práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Bakalářskou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce a konzultantem.

Datum 15. 5. 2008

Podpis

Poděkování

Rád bych poděkoval panu Ing. Pavlu Tylovi a panu Ing. Jiřímu Týřovi za odbornou pomoc a rady při tvorbě této práce.

Abstrakt

Úkolem tohoto bakalářského projektu je vytvořit editor podporující režimu WYSIWYG pro psaní dokumentů v jazyce DocBook během jejich sazby ve webovém prohlížeči. WYSIWYG editor jako součást webové stránky poběží na straně klienta. Vytvořený dokument (XML data bez formátování) je pak možné zpracovávat na straně serveru. Díky technologii XML lze použít XSL stylový jazyk a transformací převádět dokument do různých výstupních formátů. Těmi může být (X)HTML, formát pro tisk (PDF, PostScript...). Podpora režimu WYSIWYG zajistí, že uživatel nemusí znát jazyk DocBook, aby mohl editor používat. Práce se rovněž zabývá testováním vytvořeného editoru v různých webových prohlížečích.

Klíčová slova

DocBook, JavaScript, WYSIWYG editor, XSLT, W3C DOM.

Abstract

The objective of this work is to develop editor with WYSIWYG support for writing documents of DocBook document type durning it’s web browser’s composition. WYSIWYG editor as part of a web page will run on the client-side. The produced document (raw XML data) could be processed on the server-side. Thanks to XML technology the XSL stylesheet could be used to transform document to different output formats. This formats could be (X)HTML or printing format (like PDF, PostScript...). The WYSIWYG support will ensure that the user of the editor could use editor without knowing of DocBook document type. Purpose of the project is also testing the developed editor under different web browsers.

Keywords

DocBook, JavaScript, WYSIWYG editor, XSLT, W3C DOM.

Obsah

Seznam použitých zkratek

API	Application Programming Interface
CSS	Cascading Style Sheets
DB	DocBook
DHTML	Dynamic HyperText Markup Language
DOM	Document Object Model
DTD	Document Type Definition
ECMA	European Computer Manufacturers Association
FF	Mozilla FireFox
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IE	Microsoft Internet Explorer
JS	JavaScript
OOP	Objektově Orientované Programování
PDF	Portable Document Format
PHP	PHP: HyperText Preprocesor
PNG	Portable Network Graphics
RELAX NG	REgular LAnguage for XML Next Generation
PS	PostScript
RTF	Rich TextFormat
SAX	Simple Api for XML
SGML	Standard Generalized Markup Language
URL	Uniform Resource Locator
W3C	the World Wide Web Consortium
WYSIWYG	What You See Is What You Get
WWW	World Wide Web
XHTML	eXtensible HyperText Markup Language
XML	eXtensible Markup Language
XSL-FO	XSL Formatting Objects
XSLT	eXtensible Stylesheet Language Transformations
WXS	W3C XML Schema

1 Úvod

WYSIWYG je akronym anglické věty *„What you see is what you get“*, česky: „Co vidíš, dostaneš“. Tato zkratka označuje způsob editace dokumentů v počítači, při kterém je verze zobrazená v počítači vzhledově totožná s výslednou verzí dokumentu. Fráze byla původně popularizována americkým hercem a komikem Flipem Wilsonem, jehož postava *„Geraldine“* tak ospravedlňovala svoje nepředvídatelné chování. Později se tento výraz ujal v počítačové terminologii.

Nejčastěji se jako WYSIWYG editory označují textové procesory, ve kterých se editovaný text zobrazuje tak, jak bude vytištěn na papír. Příkladem editorů pracující v režimu WYSIWYG je např. Microsoft Word, či OpenOffice.org Writer, nebo WYSIWYG editory webových stránek jako jsou např. Microsoft Frontpage nebo Macromedia Dreamweaver.

Výhodou WYSIWYG editorů je, že nevyžadují od uživatele odbornost, ten může v takovém editoru psát dokumenty, aniž by rozuměl zdrojovému kódu dokumentu. Nevýhodou však je, že uživatel nemá tak velkou kontrolu nad zdrojovým kódem, který bývá objemější, než by mohl být. Z tohoto důvodu mají WYSIWYG editory řadu odpůrců, jež je označují jako editory WYSIWYNG *„What you see is what you never get“* (to, co vidíš, nikdy nedostaneš). Editory nepodoprující WYSIWYG režim se nazývají non-WYSIWYG editory.

V první časti této práce podávám stručný úvod do programování v jazyce JavaScript, ukáži jeho výhody, ale i nevýhody. Popíši problematiku tvorby WYSIWYG editoru v tomto jazyce.

Další část věnuji jazyku DocBook, technologii XML na které je založen, vysvětlím na jakém principu funguje generování různých výstupních formátů z DocBooku a ukáži pár konvenčních editorů v nichž lze DocBook poměrně dobře psát.

Následující kapitola se již bude zabývat vyvíjeným editorem. Uvedu zde, jaký postup jsem zvolil při jeho psaní, dále pak jaké možnosti editor nabízí. Uvedu příklad algoritmu, který považuji za nejzajímavější a proberu zpracování na straně serveru pro generování různých výstupních formátů.

Na závěr zhodnotím, jaké jsou vlastně možnosti pro vývoj editorů pro *„neHTML“* značkovací jazyky běžící ve webovém prohlížeči.

2 Jazyk JavaScript

2.1 Od HTML k JavaScriptu

V devadesátých letech 20. století došlo k masovému rozšiřování Internetu. V dílnách fyziků z Evropského centra pro jaderný výzkum ve Švýcarsku vznikl nápad vytvořit systém, který by umožňoval veřejně prezentovat různorodé dokumenty a jejich vzájemné logické pospojování. Tak vznikl jazyk HTML, jež pomocí HTTP protokolu umožňuje distribuci dokumentů (stránek) k uživatelům. Ten se díky komercializaci Internetu začal rozšiřovat o nové značky (*tagy*), poskytující širší možnosti formátování textu a určování pozice v rámci samotné stránky.

Samotný jazyk HTML však umožňuje vytvářet pouze stránky se *„statickým“* (neměnným) obsahem, což se v druhé polovině devadesátých let, v době, kdy Internet používá hojná veřejnost, stává nedostatečným aspektem. Vzniká tak požadavek vytváření stránek, které by poskytovaly možnost zábavy, komunikace s dalšími lidmi připojenými do Internetu, kupování zboží – požadavek určité formy interaktivity. Pro vývojáře z předních světových společnosti to znamená vývoj nových internetových technologiích, přinášejících do světa WWW nové možnosti a více dynamiky. Jednou z takových technologií je i JavaScript (JS).

2.2 Filosofie JavaScriptu a jeho možnosti

2.2.1 JavaScript není Java

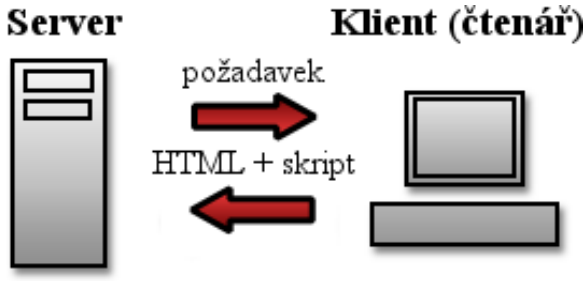
JS je často chybně zaměňován s jazykem Java. Ve skutečnosti JS, původně nazvaný LiveWire, poté LiveScript (když byl vytvořen Netscapem), by se měl spíše nazývat ECMAscript, protože tak byl přejmenován Netscapem, poté co byl standardizován asociací ECMA. Na druhé straně Java, vytvořená společností Sun Microsystems, tvoří samostatný kompilovaný programovací jazyk, s JS má pouze podobnou syntaxi (stejně jako např. jazyk C). Z toho plyne, že je Java také mnohem rozsáhlejší a komplikovanější a nabízí daleko širší možnosti. Navíc program napsaný v jazyku Java tvoří samostatný celek, zatímco program v JS je pouze součástí HTML stránky.

2.2.2 Vyčlenění jazyka

JS je objektově orientovaný vysokoúrovňový skriptovací jazyk, který je interpretovaný na straně klienta.

Objektově orientované programování je moderní programovací přístup poskytující programátorům množství účinných programovacích technik oproti klasickému sekvenčnímu programování. Opakem vysokoúrovňového programovacího jazyka je jazyk Assembler, kde každý příkaz lze přímo přeložit do strojového kódu. Skriptovací programovací jazyky se často využívají k vykonávání opakujících se úloh. Přestože mohou tvořit ucelený programovací jazyk, nejdou do takové hloubky jako komplexní programovací jazyky (absence vláken, správy paměti…). Často nevytvářejí své vlastní rozhraní, ale spoléhají na jiné programy, aby pro ně rozhraní vytvořili. V JS to např. znamená, že nemusíme prohlížeči říci kam umístit který pixel, pouze mu řekneme, že chceme změnit dokument a on to vykoná. To ale znamená, že se stejný skript nemusí v různých prohlížečích chovat stejně. Programátoři JS se snaží tento fakt co nejvíce redukovat – optimalizují své skripty. Programy napsané interpretovaným programovacím jazykem se zpracovávají v původním zdrojovém kódu tak, že se každý příkaz převádí do strojového kódu tehdy, když je volán. Naproti tomu kompilované programovací jazyky program nejprve překládají do strojového kódu a až poté je možné program vykonat. Pro programátora interpretovaného programovacího jazyka to znamená především to, že nemusí dodržovat tak striktní syntaktická pravidla a že se neprojeví chyby v částech programu, jež neproběhnou. Důležitý fakt je, že se JS interpretuje na straně klienta. To znamená, že se program nejprve nahraje do počítače, na kterým běží prohlížeč, a poté je vykonán. Opakem by byl programovací jazyk interpretovaný na straně serveru, kde klient pouze přijme výsledky (např. PHP, Perl, ASP, JSP).

Existují i jiné skriptovací programovací jazyky interpretované na straně klienta, ty však nejsou tak rozšířeny jako JS, proto se pod pojmem skript obvykle rozumí program napsaný v jazyce JS.



Obr. 2.1: Interpretace na straně klienta

2.2.3 Výhody a omezení JavaScriptu

Síla JS spočívá v jeho jednoduchosti umožňující rychlé vytváření událostí webových stránek. Programátor ani nemusí znát základní principy OOP k tomu, aby vytvořil efektivní skript obohacující jeho HTML stránky o řadu funkčních i vizuálních prvků. Mnoho JS příkazů jsou *handlery* událostí (části kódu probíhající při výskytu události jako třeba stisk klávesy), které mohou být přímo součástí HTML příkazu. Protože se JS používá k jednoduchým účelům, některé jeho možnosti zůstávají často nevyužity. Např. JS podporuje odlišení soukromého a veřejného přístupu k vlastnostím objektových tříd, vytváření podtříd a dědičnosti. V praxi se tyto techniky v JS takřka nevyužívají. Jednoduše jich není zapotřebí. Dokonce i komplikované skripty nejsou nikdy tak komplexní, aby vyžadovaly takovou úroveň řízení., ačkoliv někteří programátoři jich využívají právě proto, že jsou na to zvyklí z jiných programovacich jazyků. Je to prostě způsob, kterým jsou zvyklí psát programy.

Jak bylo řečeno, JS je interpretován na straně klienta. Díky tomu je program vykonán rychleji, než kdyby musel komunikovat se serverem. Plynou z toho ale také omezení, zužující oblast využití. Kvůli tomu nemůžeme napsat v JS program zachovávající a centrálně uchovávající údaje týkající se provozu webové stránky (např. fórum, redakční systém…). Další nevýhodou (pro tvůrce programu) je fakt, že uživatel může skript (jež může běžet pouze v prohlížeči) snadno zakázat. Proto zkušenější webmasteři vyvíjejí nejprve verze stránky bez JS a poté do něj postupně implementují skripty, které proběhnou, je-li to možné tak, aby nijak neomezovali návštěvníka stránky. Z bezpečnostních důvodů JS nedokáže přistupovat k souborům, ani nedokáže uchovávat data. Vyjímkou jsou tzv. *cookies*, které však mnoho uživatelů na svých počítačích zakazuje (právě z důvodu bezpečnosti), takže by se na ně naše stránky neměli spoléhat.

2.2.4 Využití JavaScriptu

JS se dá využít k otevírání nových oken, psaní dynamického obsahu dokumentu, kontrole údajů, jež uživatel vyplnil do formuláře a mnohému dalšímu. Nyní Vás seznámím trochu podrobněji s technikami, které jsou využity v tomto projektu.

Jednou z nejpopulárnějších aplikací JS je jeho využití pro tzv. *preload* obrázků. Zpočívá to v tom, že se veškeré obrázky načtou pomocí JS v hlavičce HTML dokumentu, ještě než se dokument zobrazí. Eliminuje se tak např. nepříjemný problém s tzv. *„rollover“* efektem, kdy se obrázek mění, nachází li se nad ním kursor myši. V případě, že takový obrázek není *„preloadován“*, může se u něj objevit časová

prodleva, před tím než se změněný obrázek zobrazí. Tuto techniku také můžeme využít, máme-li na stránce mnoho obrázků a chceme, aby se zobrazila webová stránka celá najednou a ne nejprve bez obrázků tak, že se postupně obrázky zobrazí v pořadí, jak jsou načítány.

JS si také získal veliké obliby ve spojení s *DHTML* (*dynamické HTML*). Vizuální stránku elementů HTML dokumentu určuje *CSS* (*kaskádový stylový jazyk*). DHTML by se dalo definovat jako modifikace CSS jednotlivých elementů pomocí JS. Tato technika přináší do oblasti tvorby webových stránek nové možnosti, pojem *dynamický* napovídá, že takové dokumenty se mění defakto přímo před očima uživatele. DHTML je vhodné použít, když chceme např. některý element skrýt, zobrazit, či jakkoliv změnit jeho vzhled. Neumí však přidávat nebo odebírat elementy, v takovém případě musíme sáhnout po jiné technice. Pomocí DHTML můžeme např. vytvářet *pop-up* menu, nebo tzv. *tooltipy*.

Silným nástrojem, jež JS nabízí je DOM. Jde v podstatě o objektově orientovanou reprezentaci XML nebo HTML dokumentu. DOM je API umožňující přístup či modifikaci obsahu, struktury nebo stylu dokumentu či jeho části.

Původně měl každý webový prohlížeč své vlastní specifické rozhraní k manipulaci s HTML elementy pomocí JS. Vzájemná nekompatibilita těchto rozhraní však přivedla W3C k myšlence standardizace, a tak vznikl W3C DOM. Tato specifikace je platformně a jazykově nezávislá.

DOM umožňuje přístup k dokumentu jako ke stromu. Tato technologie, nazvaná *grove* (Graph Representation Of property ValuEs), vyžaduje nahrání celého dokumentu do paměti, z čehož plyne, že její optimální použití je tam, kde je k jednotlivým elementům dokumentu přistupováno v náhodném pořadí nebo opakovaně. Existuje i alternativní technologie pro případ, že je potřeba postupná nebo jednorázová úprava – sekvenční model SAX, který má v těchto případech výhodu rychlejšího zpracování a nižší paměťové náročnosti.

Specifikace W3C DOM jsou rozděleny do několika úrovní (*DOM level*), z nichž každá obsahuje povinné a volitelné moduly. K tomu, aby nějaká aplikace mohla prohlásit, že podporuje určitý DOM level, musí tato implementovat všechny požadavky dané úrovně a všech nižších. Aplikace mohou též podporovat specifická rozšíření (anglicky vendor-specific extensions) za podmínky, že nejsou v konfliktu s W3C standardy.

- **Level 1:** Navigace v DOM (HTML a XML) dokumentu (resp. jeho stromové struktuře) manipulace s obsahem (včetně přidávání elementů). Specifické elementy HTML jsou obsaženy také.
- **Level 2:** Podpora jmenných prostorů (anglicky: XML namespace), událostí a filtrovaných pohledů.
- **Level 3:** Specifikace dále rozšiřující DOM..

2.3 JavaScript a WYSIWYG editory

WYSIWYG editory zažily v oblasti správy internetového obsahu skutečný rozmach. Tzv. *„client-side“* WYSIWYG editory tvoří důležitou součást internetových redakčních systémů, ve kterých mohou uživatelé posílat příspěvky formou HTML kódu bez znalosti jazyka HTML. Bývají napsány v JS, každý má své slabé a silné stránky.

2.3.1 Jak to funguje?

Téměř každý WYSIWYG editor napsaný v jazyce JS generuje HTML (popř XHTML) kód. Bývají založené na poměrně jednoduchém principu, a možná právě proto je jich veliké množství. Fungují tak, že na stránku vloží komponentu *iFrame*, která umožňuje *„zapouzdřit“* další HTML dokument do této komponenty. Poté skript zapne editační režim dokumentu nacházející se v *iFramu* (defakto řekne prohlížeči: *„umožni měnit uživateli obsah tohoto dokumentu“*). Ovládací prvky editoru pak volají potřebné části skriptu pro formátování dokumentu, přidávání nových elementů atp.

2.3.2 Otázka validity

Problém spočívá v tom, že HTML element *iFrame* by měl postupem času z Internetu vymizet. To znamená, že pokud chceme dodržovat standardy musíme se bez něj obejít. Jeho validní nástupce je element *Object*, který má tvořit jakýkoliv vnořený objekt na stránce (HTML stránka, ale i třeba Java applet, přehrávač záznamů...).

V tomto projektu realizuji editor pomocí *Objectu* a zhodnotím, jak dobře funguje v jednotlivých prohlížečích pro potřeby WYSIWYG editoru. Případná úskalí se pokusím vyřešit i pomocí elementu *iFrame* a výsledky porovnám.

3 DocBook

S rozvojem výpočetní techniky se rozvíjejí nejrůznější formy elektronické dokumentace. Tištěná forma se nehodí pro všechny oblasti použití. Mnoho aplikací vyžaduje krom tištěné verze dokumentu i jeho elektronickou podobu, ať už jde o publikování na webu nebo CD-ROMu.

Důležitým požadavkem pro tvorbu elektronické dokumentace je jeho generování do různých výstupních formátů (tištěná podoba, on-line nápověda, webová podoba) z jedné předlohy. Proces generování by měl být plně automatizován. Jednou z možností, jak těmto požadavkům vyhovět, je použít pro tvorbu dokumentace formát XML, což je zkratka pro *rozšiřitelný značkovací jazyk*. Pojem rozšiřitelný zde znamená, že umožňuje uživateli definovat své vlastní elementy. Tento jazyk vznikl za účelem vytvoření jednoduchého otevřeného formátu, který by nebyl úzce svázán s nějakou platformou nebo proprietární technologií. Byl vytvořen jako podmnožina obecnějšího jazyka SGML, jehož jinou aplikací je např. jazyk HTML. Protože se jedná o rozšiřitelný jazyk, má mnoho aplikací. Jednou z nejpopulárnějších je právě jazyk DocBook (DB).

3.1 Úvod do XML

Jazyk XML slouží k vyznačení částí dokumentu z hlediska jejich významu nikoliv jejich vizuální interpretace. Jednoduše můžeme mít jeden zdrojový XML dokument a připojením různých stylů z něj generovat různý výstup. Dokumenty XML jsou v textovém formátu.

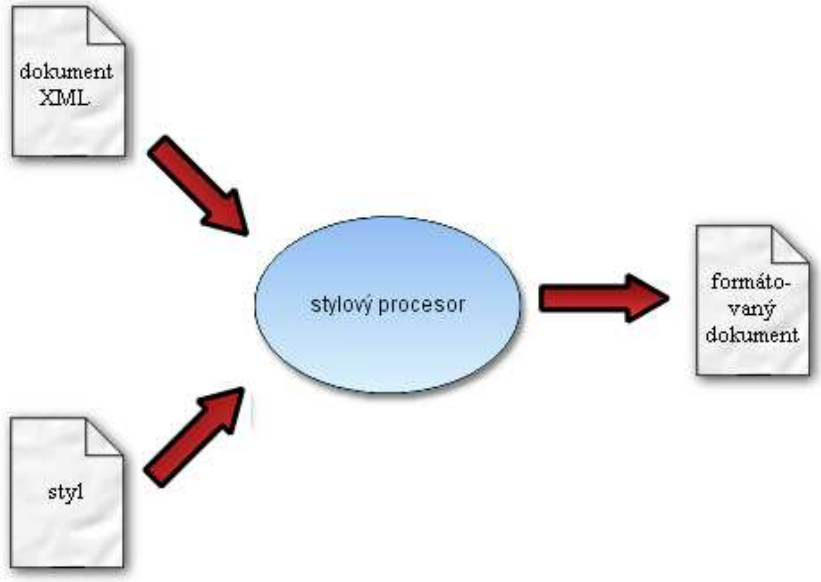
3.1.1 XML a schemata

XML dokument se skládá z elementů a jejich vlastností – *atributů*. Jaké elementy a jaké atributy může uživatel v XML dokumentu použít určuje tzv. *schematický jazyk*. Mezi ně patří DTD neboli *definice typu dokumentu*. Tento jazyk se objevil už u technologie SGML, ale postupně se ukázal být nedostatečným (nepodporuje datové typy), proto dnes není jediným schematickým jazykem. Dnes jsou např. pomocí DTD definovány jazyky HTML a XHTML. Pokud nějaký dokument odpovídá určitému schematu, říkáme, že je vůči tomuto schematu *validní*. Provádět kontrolu validity dokumentu můžeme pomocí programu zvaného *parser*. Obecně je parser program, který analyzuje předložená data (obvykle textový soubor) a vyhledá v něm prvky odpovídající definici

určitého jazyka. Obvykle provádí syntaktický rozbor zdrojových kódů. Mezi XML parsery patří např. *Xerces*.

3.1.2 Formátování XML dokumentů

Pro zobrazení XML dokumentů např. do tištěné podoby se používají tzv. *stylové jazyky*, určující vizuální stránku jednotlivých elementů. Mezi ně patří i CSS, které se však ve spojení s XML využívá čím dál méně. Jako poměrně jednoduchý jazyk nenabízí takové možnosti, jako daleko komplexnější jazyk XSL (*rozšiřitelný stylový jazyk*). Nejběžnější způsob formátování XML dokumentu je právě pomocí jazyka XSL



Obr. 3.1: Formátování XML dokumentů

3.1.3 Syntaxe XML dokumentů

Stejně jako v jazyce HTML, elementy jazyka XML jsou v dokumentu definovány pomocí *tagů*. Pod pojmem tag se rozumí část textu uzavřenou do závorek <>. V XML jsou všechny tagy párové, tzn. že každý tag musí mít i svůj uzavírací tag, který začíná lomítkem. Elementem rozumíme veškerý text uzavřený mezi dva tagy. Následuje jednoduchá ukázka XML elementu:

```
<element>Toto je obsah elementu.</element>
```

Kód 3.1: Jednoduchý XML element

Elementy mohou krom textu obsahovat i další elementy. Stejně jako v HTML i XML elementy obsahují různé atributy, záleží na použitém schematu. Syntaxe atributů je obdobná jako v HTML viz následující ukázka:

```
<element atribut = “hodnota”>
      Toto je
      <vnoreny>vnoreny</vnoreny>
      obsah elementu.
</element>
```

Kód 3.2: XML element obsahující atribut a další element

Z ukázek je patrné, že se jazyk XML syntakticky podobá jazyku HTML. V jazyce XML však platí přísnější syntaktická pravidla. Např. v XML musíme rozlišovat velká a malá písmena a žádný z tagů se nesmí křížit (před uzavřením vnějšího tagu musíme uzavřít i tag vnitřní).

3.2 Historie DocBooku

DB se zrodil v roce 1991 jako součást projektu společností HAL Computer Systems a O’Reilly & Associates jako formát určený především pro výměnu UNIXové dokumentace. Od roku 1998 je udržován společností OASIS. DB se vyvinul do formy hodící se pro technickou dokumentaci spjatou s hardwarem a softwarem, což však není omezujícím faktem, v DB se dají psát i např. knihy nebo články. Třeba dokumentace k mnoha programům je psána v DB (např. OS Linux a FreeBSD), ke skriptovacímu jazyku PHP, ke grafickým rozhraním KDE a Gnome. DB používají velká počítačová nakladatelství (jako O’Reilly) nebo velké softwarové firmy (např. Sun používá podmnožinu DB pojmenovanou SolBook).

Původně vznikl DB jako aplikace SGML jako DTD. Od verze 4 SGML DTD však začala vznikat i jeho ekvivalentní XML DTD (zachovávající označení verze). V současnosti je aktuální verze 5.0. Byla vydána poměrně nedávno a předchozí verze jsou stále dost rozšířeny. Dokumenty verze 4.x nejsou kompatibilní s verzí 5, ale mohou být konvertovány do verze 5 pomocí jazyka XSL. Verze 5 je založena na schematickém jazyku RELAX NG, ze kterého se generují ostatní formáty schémat (DTD, WXS). Změny v budoucích verzích se oznamují s předstihem, takže se na ně může uživatel dopředu připravit.

3.3 Možnosti DocBooku

Pokud se rozhodneme, že budeme vytvářet dokumentaci ve formátu DB, máme díky možnostem jazyka XML a jazyka XSL zdarma prostředek pro zcela automatické generování různých výstupních formátů z jediné předlohy. Navíc fakt, že DB existuje již řadu let dokazuje, že se tento formát osvědčil a bude se i nadále používat.

DB obsahuje řadu elementů. Mezi ně patří i elementy, které umožňují označit názvy programů, souborů, parametry příkazů, výpisy programů, obrázky, snímky obrazovky, klávesové zkratky, položky nabídek apod. Také umožňuje uživateli členit dokument do kapitol, podkapitol, sekcí a podsekcí. Získáme tak velice dobře označkovaný dokument, který se dobře převádí do dalších formátů. Používané elementy můžeme rozčlenit do dvou skupin na *blokové* elementy a *inline* elementy. Blokové elementy slouží k vyčleňení bloku dokumentu jako je např. odstavec, kapitola nebo seznam. Obvykle bývá před ním a za ním zalomená řádka, zatímco inline elementy se projevý po zformátování např. pouze změnou písma.

3.3.1 Blokové elementy

Blokové elementy jsou ty elementy, které dělí text na odstavce a podobné úseky. Mohou obsahovat i další blokové elmenty dovolí-li to použité schéma.

Asi nejběžnější blokový element v DB je odstavec. Na výběr máme tři druhy odstavců: *para*, *simpara* a *formalpara*. *Para* označuje běžný odstavec, *simpara* označuje odstavec, který může obsahovat pouze text a *formalpara* je odstavec s nadpisem. V následujícím příkladě uvádím zdrojový kód s odstavci *para* a *formalpara*.

```
<para>text odstavce</para>
<formalpara>
      <title>Nadpis odstavce</title>
      <para>další text odstavce.</para>
</formalpara>
```

Kód 3.3: Odstavce v DocBooku

Následuje ukázka jeho možného HTML výstupu, tak jak se zobrazí v prohlížeči:

Obr. 3.2: Odstavce v DocBooku – HTML výstup

Kromě možnosti členit dokument do odstavců by měl mít tvůrce dokumentu možnost vytvářet seznamy. Pro tvorbu seznamů DB nabizí mimo jiné elementy *itemizedlist* (nečíslovaný seznam) a *orderedlist* (číslovaný seznam). Opět zde uvedu příklad zdroje a HTML výstupu.

```
<itemizedlist>
      <listitem><para>položka 1</para></listitem>
      <listitem><para>položka 2</para></listitem>
</itemizedlist>
<orderedlist>
      <listitem><para>položka A</para></listitem>
      <listitem><para>položka B</para></listitem>
</orderedlist>
```

Kód 3.4: Seznamy v DocBooku

- položka 1
- položka 2

1. položka A
2. položka B

Obr. 3.3: Seznamy v DocBooku – HTML výstup

Dalším důležitým elementem jsou tabulky. Jde o poměrně komplikovaný element obsahující řadu atributů popisující danou tabulku. Následuje příklad:

```
<table>
      <title>Nadpis</title>
      <tgroup cols="2">
            <thead>
                  <row><entry>hlavička1</entry>
                  <entry>hlavička2</entry></row>
            </thead>
            <tbody>
                  <row><entry>buňka1</entry>
                  <entry>buňka2</entry></row>
                  <row><entry>buňka3</entry>
                  <entry>buňka4</entry></row>
            </tbody>
      </tgroup>
</table>
```

Kód 3.5: Tabulky v DocBooku

Nadpis tabulky

hlavička1	hlavička2
buňka1	buňka2
buňka3	buňka4

Obr. 3.4: Tabulky v DocBooku – HTML výstup

Mít dokument s bohatým obsahem textu nemusí být vždy postačující. Mnohdy potřebujeme vložit do našeho dokumentu nějaký obrázek. DB umožňuje vkládat obrázky jako samostatné bloky, nebo jako součásti textu. Tyto nároky kladu i na výsledný WYSIWYG editor. Zde nebudu uvádět příklad, pouze poznamenám, že obrázky jsou v DB tvořeny elementy *figure*, *mediaobject* a *inlinemediaobject*.

Určitě by bylo dobré, kdyby výsledný editor uměl vytvářet i odkazy. Interní odkazy odkazující na element uvnitř dokumentu realizuji pomocí elementu *link*, externí odkazy obsahující URL (adresu webové stránky) bude editor vytvářet elementem *ulink*.

3.3.2 Inline elementy

Inline elementy obvykle slouží k vyznačení významu částí textu. DB jich nabízí celou řadu a i díky nim umožňuje DB vyvíjet dobře označkované dokumenty. Otázkou je, které z nich by měl podporovat i můj editor a jakou formou.

3.3.3 Shrnutí

Jazyk DB je poměrně obsáhlý. Proto je při návrhu editoru důležité dobře zvážit, které jeho možnosti bude ovládat i výsledný editor. Editor by měl implementovat možnost snadného rozšíření tak, aby si každý jeho uživatel mohl editor snadno rozšířit o jím používané elementy.

3.4 Jazyk XSLT

Výhodou DB (resp. XML) oproti klasickému HTML je podpora stylového jazyka XSL. Je to vlastně rodina jazyků, určující, jak se mají XML soubory formátovat a převádět do jiných formátů. Zahrnuje tři jazyky:

- **XPath**: jazyk k adresování částí XML dokumentu (je používán např. v XSLT).
- **XSL-FO**: XML jazyk pro specifikaci vizuálního formátování XML dokumentů. Jeho základní myšlenkou je, že uživatel nepíše dokumenty přímo v XSL-FO, ale v jakémkoliv jiném XML jazyce (jako je i DB). Poté dokument převede pomocí XSLT předpisu na formátovací objekty. Ty jsou interpretovány FO procesorem

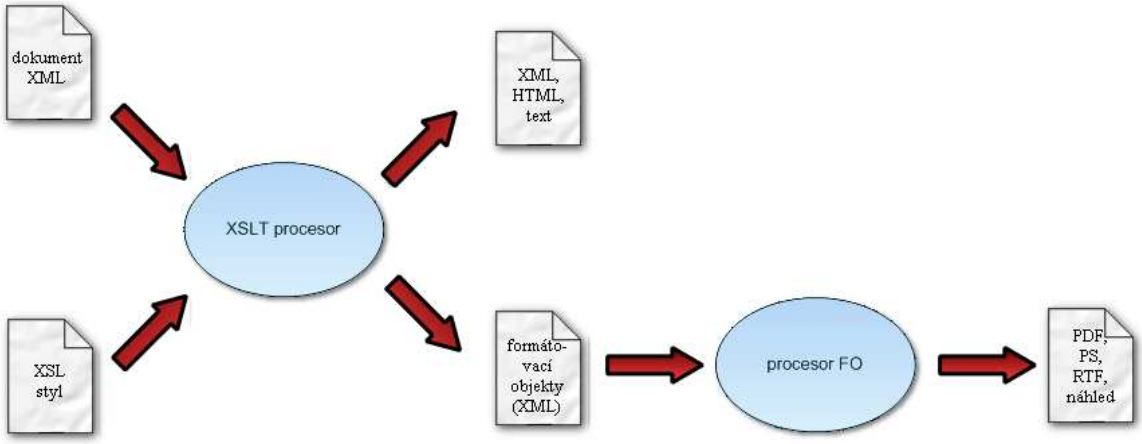
do formátu, který je čitelný, tisknutelný, či obojí. Nejobvyklejšími výstupními formáty FO procesorů jsou PDF a PostScript, ale můžete se setkat i s jinými typy výstupů, (jako například RTF či dokonce přímé grafické zobrazení do okna na displeji, viz. obr. 3.5). Podpora FO je daleko vzácnější narozdíl od XSLT. Navíc existující implementace nebývají úplné. Známým FO procesorem je FOP vyvíjený společností Apache, zvládající velikou část jazyka XSL-FO.

- **XSLT (transformační jazyk XSL)**: jazyk k převádění dokumentů tvořící stěžejní součást jazyka XSL, budu se jím zabývat podrobněji.

Specifikace těchto tří jazyků jsou dostupné ve formě W3C doporučení.

3.4.1 Princip transformace

Transformace XSLT slouží k převodům zdrojových dat ve formátu XML do libovolného jiného požadovaného formátů. Tímto formátem nejčastěji bývá HTML, může jím být i XML formát nebo formát libovolných jiných datových struktur. Provádějí se pomocí *XSLT procesoru.* Pro provedení transformace je potřeba zdrojový XML soubor a XML soubor tvořící XSL styl. Formáty pro tisk se vytváří použitím specializované aplikace zvané formátovací procesor (*FO procesor*).



Obr. 3.5: Formátování pomocí XSLT

Základ XSL stylu tvoří tělo šablony. Definuje přesně, jak se části transformovaného dokumentu budou zpracovávat. Může obsahovat přímo elementy z výsledného dokumentu (např. HTML tagy v případe transformace do HTML). Části transformovaného dokumentu jsou vybírány výrazem pomocí jazyka XPath. Šablona tedy vytváří transformační pravidla a podmínky pro jejich aplikaci.

Struktura výstupu XSLT není definována přímo standardem a je závislá na použitém XSLT procesoru. Procesory bývají přímo součástí programovacích jazyků

(např. Java, C#, PHP) nebo tvoří knihovny (např. Xalan, Saxon). Pracují tak, že nejprve nahrají celý XML dokument do paměti a vytvoří si jeho stromovou reprezentaci obdobně jako v případě DOMu (viz. kapitola 2.2.4). Tento strom je pak postupně procházen od kořene v pořadí, v jakém jsou elementy obsaženy v dokumentu (tzv. *průchod do hloubky*). V okamžiku, kdy je nalezena šablona odpovídající uzlu ve stromu, začne vypisovat svůj obsah na výstup.

XML dokument je možné transformovat rovněž na straně klienta pomocí webového prohlížeče, který má v sobě integrovaný *parser*. Nejčastějším výstupním formátem je HTML. Téměř všechny přední prohlížeče podporují XML a XSLT. Jsou jimi např.

- **Internet Explorer**: používá parser MSXML (verze 6 podporuje formátování XML pomocí CSS i XSLT, verze 5 není kompatibilní s oficiálním W3C XSL doporučením).
- **Netscape/Mozila Firefox**: pužívá parser Expat (podporuje XML, CSS i XSLT).
- **Opera**: verze 9 podporuje XSLT zatímco verze 8 pouze CSS.

3.4.2 Vlastnosti XSLT

Zdrojová data pro transformaci mohou obsahovat libovolné znakové sady. Ve zdrojovém dokumentu však musí být v záhlaví označeno kódování znaků.

Protože samotný XSL styl je v podstatě zvláštním druhem XML dokumentu, tvoří jej klasický textový soubor. Ten je čitelný bez užití specializovaného editoru. Díky tomuto principu je možné tyto soubory snadno generovat pomocí počítačových programů. Příkladem může být získání dat z databáze (či jiných datových struktur), jejich snadná konverze do XML a následná aplikace XSLT například pro převod do HTML. Navíc je XSLT nezávislé na použité platformě.

Z matematického hlediska není XSLT jednoznačné, není ani symetrickou funkcí. Pro některé procesory však existuje alespoň jeden případ $T(x,y) = x$, kde x je zdroj a y je vzorec transformace. Neexistuje však případ $T(x,y) = y$.

3.4.3 Modifikace stylů

Uživatel si může vybrat z řady dostupných stylů. Nemusí tedy psát své vlastní, stačí upravit nějaký již existující. Norman Walsh a DocBook Development Team vyvíjí řadu XSL stylů pro DB, které jsou dostupné zadarmo. Tvůrce DB dokumentů si tak může takový styl upravit pro potřeby své aplikace. Např. chceme vytvořit HTML

dokumentaci pro nějaký softwarový produkt. Samotný DB umožní vyznačit v zdrojovém dokumentu např. klávesové zkratky, položky nabídek apod. Takto označkovaný dokument tvoří dobrý základ pro jeho formátování. Stačí pak vybrat nějaký DB XSL styl generující HTML a upravit ho tak, aby lépe splňoval naše potřeby. Chování stylů lze ovlivňovat pomocí parametrů, složitější úpravy se provádějí přímo modifikací XSL kódu.

3.5 Podpora DocBooku v editorech.

Pro editaci DB (resp. XML) dokumentů stačí jednoduchý textový editor. Nicméně pro pohodlné a efektivní vytváření a úpravu DB dokumentů samozřejmně potřebujeme editor, který nám práci maximálně usnadní. Mnoho XML editorů dnes obsahuje podporu DB včetně stylů pro konverzi DB do výstupních formátů.

3.5.1 XML Mind

XML Mind je WYSIWYG editor XML, který je pro osobní použití zdarma. Dodáva se s předkonfigurovanou podporou schémat a stylů pro DB. Umožňuje validaci dokumentu a díky tomu, že je napsán v jazyku Java, existuje na mnoha platformách. Placená verze umožňuje i snadné spouštění konverze DB do různých výstupních formátů.

3.5.2 Epic

Epic je jeden z nejlepších WYSIWYG XML editorů na trhu. Má předkonfigurovatelnou podporu DB a v případě dokoupení publikačních modulů umí editor sám formátovat DB dokumenty.

4 Editor a jeho vývoj

Stěžejní součástí tohoto projektu je samotný editor. Zvolil jsem pro něj pracovní název WYSID, což je zkratka *What You See Is Docbook* (to, co vidíte, je Docbook).

4.1 Postup

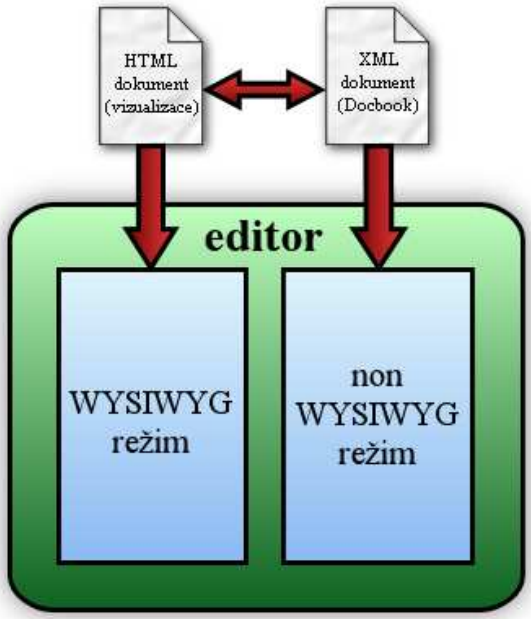
Dokument napsaný v jazyce DB je sám o sobě pouze holý XML dokument bez formátování určujícího jeho vzhled. Pro vytvoření WYSIWYG editoru pro DB je tedy třeba dokument nějak vizualizovat. Samotný editor běží ve webovém prohlížeči, vizualizace tedy probíhá formou HTML kódu. Znamená to, že editor pracuje se dvěma dokumenty – zdrojovým XML dokumentem představujícím DB a HTML dokumentem představujícím jeho vizualizaci.

Editory napsané v JS běží ve webovém prohlížeči a bývají využívány návštěvníky stránky. Podpora režimu WYSIWYG je zde důležitá, protože návštěvník nemusí znát značkovací jazyk, ve kterém editor vytváří dokumenty. V případě, že uživatel daný značkovací jazyk ovládá, může editor přepnout do režimu editace zdrojového kódu (režim non-WYSIWIG), aby tak například doplnil do dokumentu značky (popř. atributy), které editor ve WYSIWYG režimu neovládá.

V případě klientských HTML editorů se o vizualizaci HTML kódu stará prohlížeč. Prohlížeč webových stránek samozřejmě ví, jak se mají které HTML značky zobrazovat. S podporou formátování XML dokumentů pomocí XSL stylů je ale situace horší. Navíc v případě HTML editoru má programátor k dispozici rozhraní prohlížeče určené pro vývoj takového editoru.

Základní obtíže tohoto projektu tedy jsou:

- Jak převádět DB do HTML (na straně klienta)?
- Jak zajistit vazbu mezi zdrojovým DB a jeho HTML vizualizací, aby změna jednoho ovlivnila druhý? Tento problém jsem vystihl na následujícím obrázku:



Obr. 4.1: DocBook a jeho vizualizace

Ve svém editoru potřebuji převádět DB do HTML. Standardně se pro tento převod využívají XSLT (viz. kapitola 3.4) a k tomuto účelu jsou k dispozici zdarma příslušné styly. Protože úkolem tohoto projektu je vytvořit editor běžící na straně klienta, je potřeba transformace provádět přímo v prohlížeči. Odpadá tak sice potřeba komunikovat se serverem, ale naopak vzniká závislost na prohlížeči a jeho podpoře tyto transformace provádět. Další věc je, zda se vůbec prohlížeč dokáže vypořádat s tak propracovanými šablonami, které jsou vyvíjeny pro rozsáhlý jazyk DB.

Jinou méně obvyklou možností jak převádět XML dokumenty do HTML podoby, je prostřednictvím programového kódu. Pomocí rozhraní DOM lze snadno procházet XML dokumentem a během tohoto průchodu generovat výsledný dokument. Protože má dokument napsaný v DB obdobnou strukturu jako odpovídající HTML dokument, algoritmus není příliž komplikovaný. Odpadá tak potřeba podpory XSLT v prohlížeči, proto jsem zvolil tuto metodu.

Podstatně komplikovanějším a stěžejním problémem je, jak provázat Docbook s jeho vizualizací tak, aby si vzájemně odpovídali při změně jednoho či druhého.

Protože je vizualizace HTML kód, využil jsem pro její editaci již dříve zmíněné rozhraní prohlížeče – rozhraní používané pro vývoj klientských HTML WYSIWYG editorů. Vývojáři různých webových prohlížečů mají evidentně odlišný vkus a představu o tom, jak by se měl WYSIWYG editor chovat. Neexistuje žádný standard definující, jak by měl editor reagovat na situace kdy je více možností, takže v každém prohlížeči se chová editor trochu jinak.

Jak tedy zajistit, aby se při editaci vizualizace změnil i zdrojový DB? Zkoušel jsem provázat HTML elementy s elementy DB pomocí JS. Při editaci textu HTML elementu se tak změnil i text příslušného elementu DB. Chtěl jsem skript rozvinout tak, aby byl schopen reagovat i na další změny jako je např. přidávání a mazání elementů nebo formátování seznamů. Protože se rozhraní WYSIWYG chová jinak v různých prohlížečích, plyne z toho potřeba rozsáhlé optimalizace skriptu. Některé situace mají navíc vedlejší efekty. Pokud např. editujeme prázdnou položku seznamu a stiskneme *enter*, místo přidání nové položky proběhne roztržení seznamu na dva nové seznamy tak, že místo prázné položky je vytvořen nový odstavec mezi těmito seznami. Místo předpokládané akce – vložení nové položky seznamu tedy proběhne něco úplně jiného a skript by to měl rozpoznat a správně na to reagovat. Možností je více, uživatel může např. vybírat elementy a přesouvat je metodou *drag and drop* (tzv. metoda *„chyť a pusť*“), využívat schránku atd. V rámci těchto operací se může třeba prohlížeč rozhodnout, že ve výsledku spojí dva seznamy za sebou v jeden. Počáteční nadšení z faktu, že o editaci vizualizace se postará prohlížeč tedy padá, protože je potřeba na něj bedlivě dohlížet a kopírovat jeho chování. Výhoda, že prohlížeče v sobě implementují rozhraní pro WYSIWYG editory tak doplácí na nevýhody spjaté s optimalizací a špatnou kontrolu nad záležitostmi, které si prohlížeč řeší po svém.

Nakonec jsem se rozhodl, že zkusím jiný postup. Místo toho abych napodoboval chování prohlížeče a obtížně zjišťoval, co se vlastně při jakém příkazu uživatele s dokumentem stalo, nechám vytvořit prohlížeč HTML dokument, a na základě výsledku skript určí odpovídající dokument formátu DB.

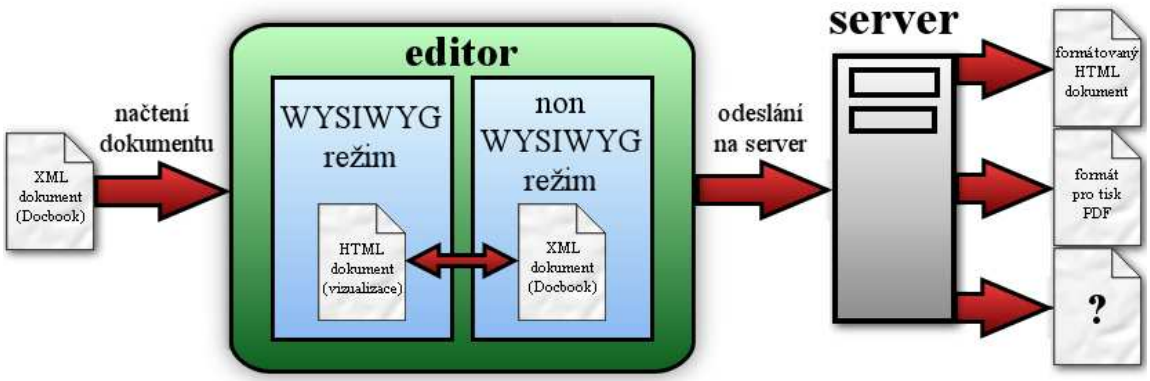
4.2 Návrh editoru

Editor by měl zvládat následující operace:

- načtení dokumentu formátu DB,

- editaci tohoto dokumentu buď úpravou zdrojových dat formátu DB, nebo editací jeho vizualizovaných HTML dat WYSIWYG formou,
- odeslání dokumentu formátu DB na server pro následný převod pomocí XSLT (viz. kapitola 3.4) do výstupního formátu (např. HTML, PDF…).

Uživateli editoru, který nezná jazyk DB se tedy může editor ve WYSIWYG režimu jevit jako klasický HTML editor, který ovládá více výstupních formátů krom formátu HTML. Tvůrci stránek (webmasterovi) implementujícímu tento editor na své stránky se však nabízí širší možnosti zpracování dokumentu na serveru. Ten díky tomu, že na server je posílán XML dokument může úpravou stylů usměrňovat výsledný formátovaný dokument pro potřeby své aplikace.



Obr. 4.2: Princip editoru

4.2.1 Elementy podporované editorem

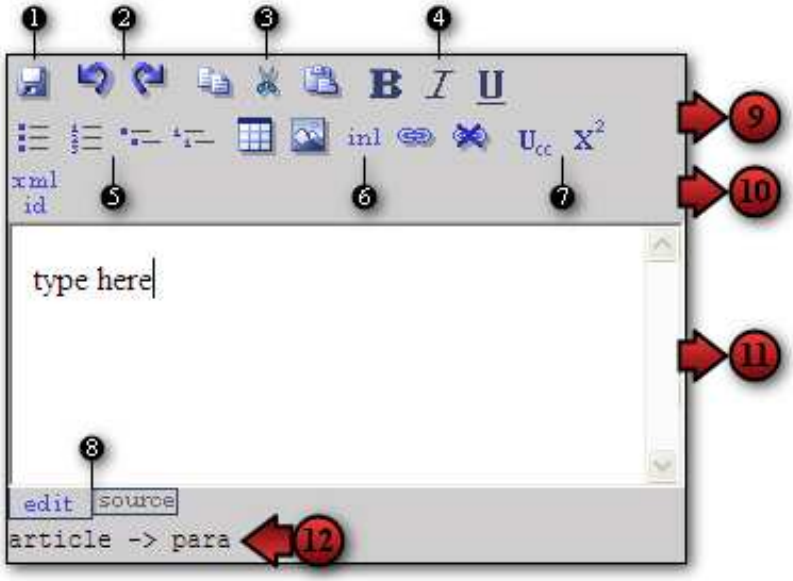
Do editoru WYSID jsem implementoval základní funkce pro značkování textu, jako jsou např. seznamy, tabulky, obrázky a odkazy. Některé z nich lze programovat přímo pomocí příkazů, které ovládá sám prohlížeč, pro vývoj propracovaného editoru to však zdaleka nestačí. Lze tak např. formátovat text do seznamů, nestačí to ale pro volbu jejich číslování nebo tvorbu víceúrovňových seznamů. Protože jsem chtěl aby můj editor disponoval i těmito vlastnostmi, implementoval jsem je do něj pomocí rozhraní W3C DOM (viz. kapitola 2.2.4). Následuje výčet možností podporovaných přímo editorem WYSID ve WYSIWYG režimu:

- formátování *tučné*, *kurzíva*, *podtržené*,
- členění textu do odstavců a víceúrovňových seznamů s volbou číslování,
- vkládání tabulek a obrázků (u tabulek možnost volby ohraničení a u obrázků možnost volit mezi obrázkem vloženém do textu nebo obrázkem tvořící samostatný blok),

- označkovat text jako jednoduchý *inline* element po výběru z nabídky členěné do kategorií, tyto kategorie zahrnují: bibliografické elementy (*bibliography*), elementy značící chybu programu (*error*), součást grafického rozhraní (*GUI*), prvky klávesnice (*keyboard*), operačního systému (*operating system*), značkovací elementy (*markup*), elementy označující produkt (*product*), součásti programového kódu (*programming*) a technické elementy (*technical*),
- vkládání hypertextových odkazů a odkazů na elementy v dokumentu,
- formátování horních a dolních indexů.

4.2.2 Grafické rozhraní

Grafické rozhraní (GUI) je druh uživatelského rozhraní pro interakci uživatele s počítačovým programem. U klientských WYSIWYG editorů je navrženo tak, aby bylo kompaktní, a editor tak nemusel zabírat mnoho místa na stránce. Slouží pro editaci dokumentu ve WYSIWYG režimu.



Obr. 4.3: GUI editoru WYSID

Na obrázku 4.3 je ukázka WYSIWYG režimu. Jednotlivé části jsou zde očíslovány číslicemi 1 až 12. Úplně nahoře je ovládací panel (9 a 10). Jeho hlavní část (9) obsahuje základní funkce pro práci s dokumentem a jeho formátování (1–7). Poslední řádek panelu (10) slouží k editaci atributů a dynamicky se aktualizuje podle právě vybraného elementu. Pod ovládacím panelem je editační pole (11). Dole se nachází tzv. *status bar* (12), ve kterém se zobrazuje značka (*tag*) právě vybraného elementu s jeho rodiči.

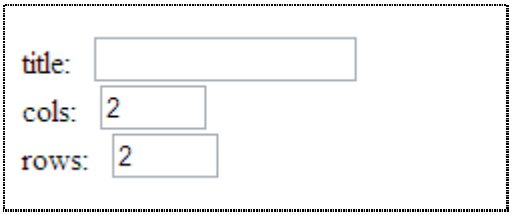
Prvky ovládacího panelu mají ikonový charakter. Jednotlivé ikony reprezentují tlačítka. To první (1) slouží k uložení dokumentu. Další (2 a 3) jsou tlačítka pro funkce

zpět a *vpřed*. Pracovat se schránkou lze pomocí klávesových zkratek *CTRL+C* (kopírovat), *CTRL+X* (vyjmout) a *CTRL+V* (vložit) nebo pomocí tlačítek (3). Další tlačítka jsou pro formátování *tučné*, *kurzíva* a *podtržené* (4). Následují tlačítka pro členění seznamů (5), vkládání tabulek, obrázků, formátování *inline* elementů a odkazů (6) a formátování horních a dolních indexů (7). Poslední ovládací prvek (8) slouží pro přepínání mezi WYSIWYG a non-WYSIWYG režimem.

Většina tlačítek má charakter zamáčknout/vymáčknout, jiná po kliknutí zobrazí formulář, který je potřeba vyplnit pro vložení nového elementu.

4.2.3 Vkládání tabulek

Jedním ze složitějších elementů jsou tabulky. Tabulka je do dokumentu vložena po vyplnění formuláře na následujícím obrázku:

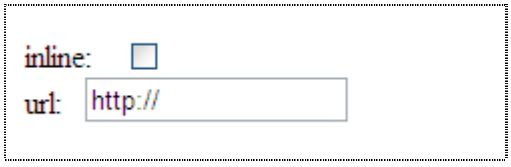


Obr. 4.4: Vkládání tabulek

Do kolonky *title* se vyplňuje nadpis, do kolonek *cols* a *rows* počet sloupců a řádků nové tabulky.

4.2.4 Vkládání obrázků

Pro vložení obrázku je potřeba vyplnit následující formulář:



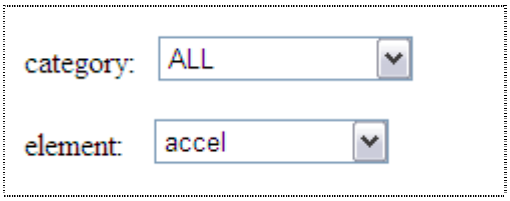
Obr. 4.5: Vkládání obrázků

Zaškrtávací kolonka *inline* určuje, zda bude obrázek tvořit součást textu (inline element) nebo tvořit samostatný blok (jako blokový element). V kolonce URL se vyplňuje adresa obrázku.

4.2.5 Formátování inline elementů

DB disponuje řadou *inline* elementů (viz. kapitola 4.4.2). Jsou jednoduché a slouží k vyznačení významu částí textu. Protože je jich veliké množství, rozčlenil jsem je do kategorií, tak jak jsou uvedeny v publikaci [1]. Pro jejich formátování v editoru WYSID

stačí vybrat text, který chceme zformátovat, a po stisku příslušného tlačítka vyplnit formulář.

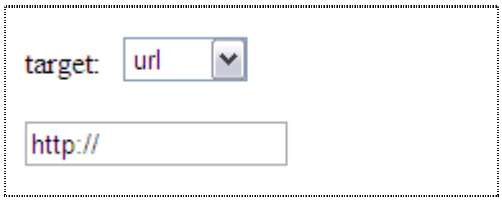


Obr. 4.6: Formátování inline elementů

Ten obsahuje dvě roletová menu (viz. obr 4.6). V tom prvním se vybírá kategorie (je možno vybrat všechny tak, jak je to na obrázku) a během výběru kategorie se dynamicky aktualizuje druhé s výběrem konkrétních elementů.

4.2.6 Formátování odkazů

Formulář pro formát odkazu vypadá následovně:



Obr. 4.7: Formátování odkazů

I zde (stejně jako u inline elementů) je třeba nejprve vybrat text. Roletové menu označené zde jako *target* určuje, zda se jedná o hypertextový odkaz (cíl URL adresa) nebo o odkaz na element uvnitř dokumentu (cíl atribut *xml:id*).

4.2.7 Editace atributů

Panel pro editaci atributů slouží ke snadnému nastavení atributů podporovaných editorem. Každému elementu můžeme nastavit atribut *xml:id* (sloužící jako unikátní identifikátor xml elementu), proto ikona pro jeho editaci je přitomna vždy. Pokud je vybrána položka seznamu, nachází se zde ikony pro výběr číslování v případě číslovaných seznamů a pro výběr značek u nečíslovaných seznamů. U tabulek lze nastavovat atribut pro volbu ohraničení tabulky a oddělovač sloupců a řádků.

4.3 Skript

Editor je napsaný v jazyce JS (viz. kapitola 2) a z veliké části se opírá o rozhraní W3C DOM (viz. kapitola 2.2.4).

4.3.1 Lehký úvod do objektově orientovaného programování

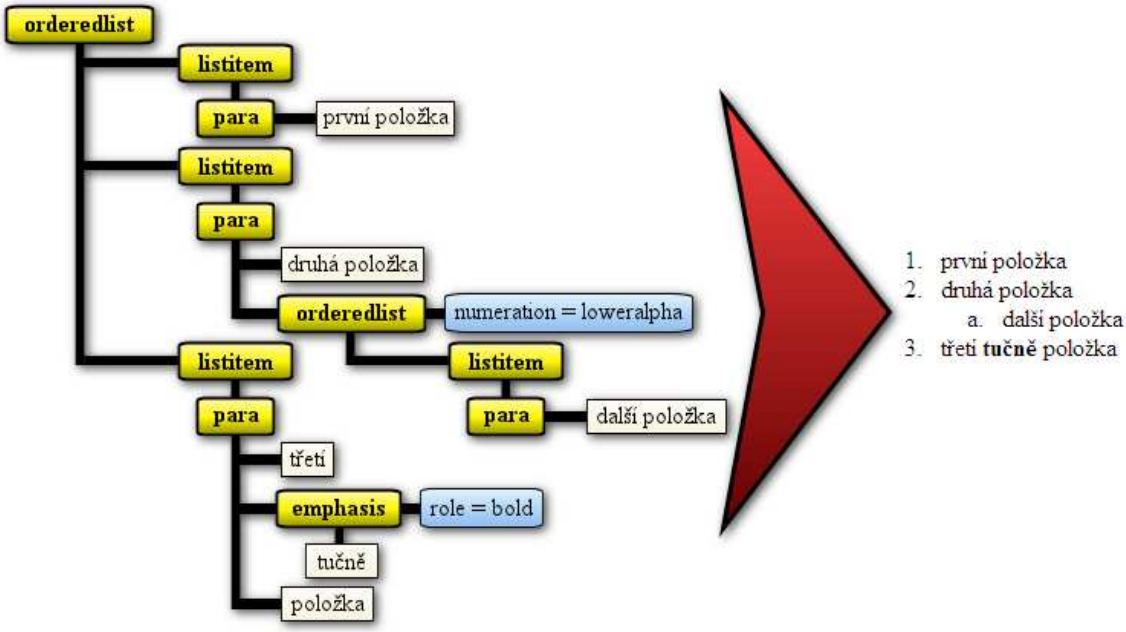
Objektově orientované programování (OOP) je programovací přístup podporovaný moderními programovacími jazyky (např. C++, C#, Java, ale i JS). Opakem OOP je sekvenční programování a JS (stejně jako C++) podporuje oba přístupy.

Základní vlastnosti OOP jsou *zapouzdření*, *dědičnost* a *polymorfismus*. V JS se používají zřídkakdy a nacházejí své uplatnění především u robustních aplikací, proto se jimi nebudu zabývat. Nicméně samotná možnost členění programu na objekty je velmi užitečná. V OOP je program tvořen objekty tak, že každý řeší vymezenou část problému. Ty si mohou navzájem vyměňovat informace, ale jinak fungují jako samostatné celky řešící určitou úlohu. Modifikace programu se tak stávají daleko pohodlnější, protože probíhají na úrovni jednotlivých objektů nezávisle na zbytku programu. Pokud je tedy třeba použít jinou myšlenku k dosažení stejného cíle, stačí přepsat příslušný objekt, ne celý program. Takový program je navíc přehledný (pokud je objektový model šikovně navržen), a v případě, že se jedná třeba o knihovnu, padá potřeba složité dokumentace. Navíc je pružný a díky tomuto faktu jsem nemusel přepisovat celý program, když jsem zkoušel různé postupy. Přestože jsem prováděl několikrát rapidní změny v kódu, nestalo se mi, že by program nebo jeho část náhle přestala fungovat. Naopak jsem byl i mile překvapen, když stejný kód fungoval lépe než jsem čekal, i když se jádro editoru takřka celé změnilo.

Objekty by se daly chápat jako speciální strukturované datové typy, ve kterých můžeme zapouzdřovat libovolné jiné datové typy (proměnné, funkce, jiné objekty). Alokovaný objekt v paměti počítače se nazývá jeho *instance* a vytváří se pomocí zvláštní funkce bez návratové hodnoty zvané *konstruktor*. Funkce tvořící součásti objektu jsou jeho *metody*, mívají za úkol vykonat určitou akci popř. vrátit určitou informaci. Proměnné existující v rámci objektu jsou jeho *vlastnosti*.

4.3.2 DOM Level 1 a stromová reprezentace dokumentu

Základ specifikace W3C DOM tvoří jeho první úroveň, která umožňuje procházení stromu představující dokument a manipulaci s jeho obsahem. Vyšší úrovně bývají u prohlížečů podporovány zřídkakdy. Pro lepší představu co je míněno pod pojmem stromová reprezentace dokumentu uvádím následující obrázek:



Obr. 4.8: Obraz dokumentu parsovaného do DOMu

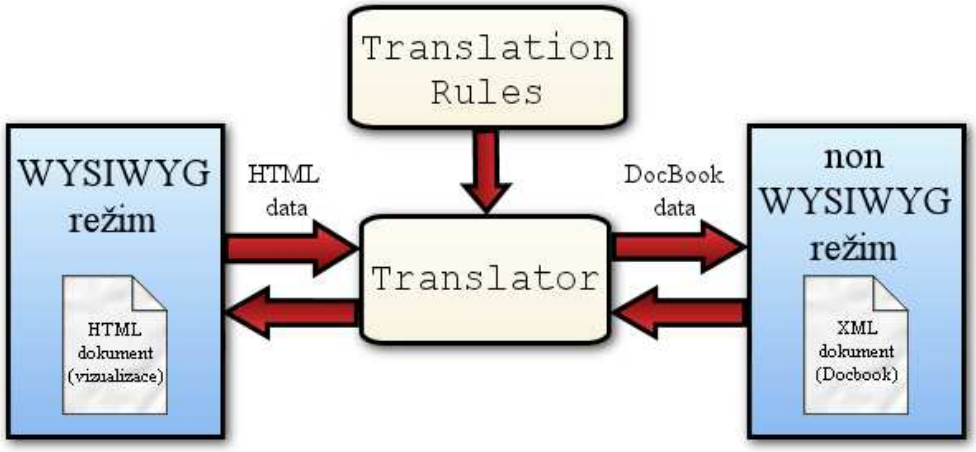
Je na něm uveden seznam napsaný v jazyce DB včetně jeho vizualizace (vpravo). Parserováním do DOMu vlastně vytvoříme jeho obraz v paměti počítače, ve kterém jsou jednotlivé větve navázány na sebe. Každá větev je objekt určitého typu. V ukázce na obrázku 4.8 jsou větve *orderedlist*, *listitem*, *para* a *emphasis* větve typu elemenent, větve *numeration* a *role* jsou typu atribut a větve *první položka*, *druhá položka*… jsou textové řetězce. Takový strom je možné procházet libovolným směrem – od kořene do hloubky i zpět. Úplný kořen stromu je objekt typu *document*, který umožňuje vytváření instancí nových větví. Ty je pak možné připojit k libovolné větvi typu element.

Stromová reprezentace je výstižná pro značkovací jazyky, proto součástí non-WYSIWYG editorů bývá i okno s rozbalovatelným stromem pro orientaci v dokumentu a modifikaci jeho struktury.

4.3.3 Objektový model

Problematiku editoru jsem rozčlenil do sady objektů. Základním objektem je `Wysid`, ten představuje samotný editor. Zapouzdřuje v sobě všechny ostatní objekty a pomocí rozhraní prohlížeče a DOM zprostředkovává WYSIWYG režim editoru. K tomu využívá některé další objekty (např. objekty tvořící ovládací prvky uživatelského rozhraní nebo objekt realizující *status bar*).

Důležitou součástí objektu `Wysid` je objekt `Docbook`. Ten v sobě uchovává data formátu DB a umožňuje jejich načítaní pomocí metod. Umí tato data parserovat do DOMu pro jejich zpracování pomocí programového kódu a převádět nazpět do formátu XML (tzv. *„serializovat"*). Objekt `Wysid` tedy řeší editaci HTML vizualizace – WYSIWYG režim editoru a jeho dceřinný objekt `Docbook` umožňuje non-WYSIWYG režim. Samy o sobě by však nevěděly jak se mají měnit v závislosti na změnách jejich protějšku. Při načtení dat formátu DB je nutné je vizualizovat pro WYSIWYG režim a během editace v tomto režimu je třeba aktualizovat zdrojový kód formátu DB (data držena objektem `Docbook`). V kapitole 4.1 jsem pojednal o tom, že jsem nakonec upustil od původní myšlenky promítání změn provedených ve WYSIWYG režimu do DB. Rozhodl jsem se tedy generovat DB z HTML kódu pomocí skriptu. Tuto problematiku řeší objekt, který jsem nazval `Translator`.



Obr. 4.9: Funkce objektu Translator

Také jsem uvedl, že pro vizualizaci DB využiji skript. Ten funguje na základě průchodu stromu zdrojového dokumentu od kořene do hloubky a paralelně generuje výsledný dokument. Je tedy potřeba převody provádět obousměrně (viz. obr. 4.9). Je zbytečné psát dva algoritmy pro dva druhy převodů tam a zpět, oba převody mohou existovat v rámci stejného algoritmu. Je však důležité, aby během těchto převodů nedocházelo ke

ztrátě dat. Uživatel může v non-WYSIWIG režimu přidat do zdrojového DB nějaký element (resp. atribut), který editor neumí vizualizovat, při zpětném převodu se tyto elementy (resp. atributy) nesmí ztratit. Objekt `Translator` provádí obousměrné převody na základě jedněch převodních pravidel. Ta jsou definována v jeho dceřinném objektu `TranslationRules`. V případě, kdy chceme editor rozšířit o nový element, stačí pomocí metody objektu `TranslationRules` definovat, jak se daný element převádí.

4.3.4 Převodní algoritmus

Nyní trochu nastíním, jak jsem postupoval při vývoji algoritmu tvořící zásadní součást objektu `Translator`.

Aby se celý dokument převedl, je nutné celý ho projít od kořene do hloubky. Pro tento průchod jsem využil specifikace uvedené v rozhraní W3C DOM Level 1 a programovací techniku rekurze.

Rekurze spočívá v opakovaném voláním funkce sebe sama, z toho také vyplývá základní pravidlo pro psaní rekurzí: každá rekurze musí být konečná. Pokud by se stalo, že by nějaká funkce v programu neustále rekurzivně volala sama sebe, program by nakonec spotřeboval všechnu paměť počítače a nakonec by zkolaboval. Průchod od kořene do hloubky připomíná tzv. *„rozlévací“* algoritmus – ten jako by se postupně roztekl celým stromem do všech větví. Pro naprogramování rozlévacích algoritmů je možno použít právě rekurze. Dokument samozřejmně nemůže být nekonečně veliký, takže rekurzivní rozlévání stromem dokumentu je vždy konečné.

```
function translate(node)
{
      for (var i = 0; node.childNodes[i]; i++) {
            if (node.childNodes[i].firstChild)
                  {translate(node.childNodes[i])}
      }
}
```

Kód 4.1: Průchod do hloubky parametru node

V ukázce kódu 4.1 je funkce `process` se vstupním parametrem `node`. Aby funkce proběhla pro všechny elementy obsažené v dokumentu, stačí ji zavolat s parametrem kořenového (*root*) elementu. V těle funkce je cyklus `for`, který zajistí, že následující příkaz bude proveden pro všechny dceřinné větve parametru `node`. Tím je podmínka `if`. Ta se zeptá, zda momentální dceřinná větev `node.childNodes[i]` větve `node`

obsahuje další dceřinné větve. Jestliže ano, je volána znovu funkce `process`, tentokrát je však jako parametr `node` použita větev `node.childNodes[i]`. Ukázka neudělá nic viditelného, pouze proběhne pro všechny elementy vstupního parametru.

Aby algoritmus z předchozí ukázky mohl paralelně generovat větve převedeného dokumentu, je nutno ho rozšířit o další parametr. Bude jím odkaz na element do kterého se připojí převedená větev, nazvěme ho `result` viz. následující ukázka:

```
function translate(node,result)
{
      for (var i = 0; node.childNodes[i]; i++) {
            if (node.childNodes[i].firstChild)
                  {translate(node.childNodes[i],translateNode(node.chi
      ldNodes[i], result))}
            else
                  {translateNode(node.childNodes[i],result);}
      }
}
```

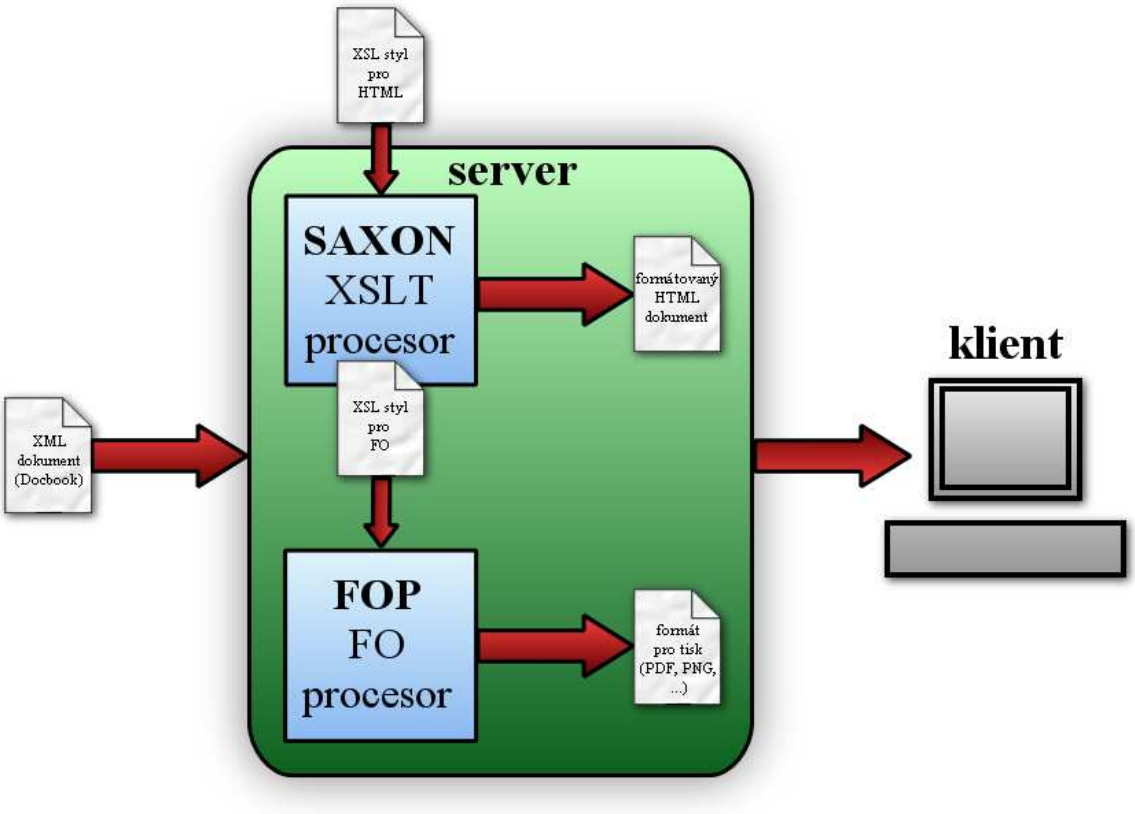
Kód 4.2: Základní konstrukce převodního algoritmu

Funkce `translateNode` provádí převod jednotlivé větve. Na základě informací z prvního parametru vytvoří novou větev výsledného dokumentu a tu připojí k větvi, na kterou odkazuje druhý parametr. Jestliže se větev `node` dále nevětví, není třeba rekurzivně volat funkci `translate`, ale i tento element je třeba převést, proto je volána funkce `translateNode` s příslušnými parametry (kód následující po `else`). Jelikož tato funkce vrací odkaz na místo kam připojit příští výslednou větev, je použita jako druhý parametr rekurzivního volání funkce `translate`. Převod jednotlivé větve funkcí `translateNode` je pak otázkou její řídící logiky.

4.3.5 Výstupní formáty a transformace na straně serveru

Pro uložení dokumentu do souboru je třeba poslat jeho zdrojový kód na server. K tomuto účelu jsem vytvořil skript v jazyce PHP, což je zkratka *PHP: Hypertextový Preprocesor*.

Skript umožňuje generovat různé výstupní formáty pomocí XSLT (viz. kapitola 3.4). Výsledek je poslán zpět klientovi (viz. obr. 4.10).



Obr. 4.10: Transformace XSLT na straně serveru

Ještě před odesláním na server je potřeba vyplnit formulář, ve kterém uživatel vybere výstupní formát. Zde je možné vybrat formát XML. V takovém případě neproběhne žádná transformace a ze serveru je vrácen přenos XML souboru, jehož obsah tvoří zdrojový kód DB.

Pro transformace do HTML formátu editor používá XSLT procesor *Saxon* verze 6.5.5. Je to Open-Source projekt napsaný v Javě a jeho autorem je Michael Kay. Formát pro tisk zajišťuje procesor formátovacích objektů *FOP* verze 0.95beta vydaný společenstvím Apache. Jako styly jsem použil XSL styl generující (X)HTML pro Saxon a XSL styl generující formátovací objekty (FO) pro FOP. Jejich autor je Norman Walsh. Mezi formáty, které umí generovat FOP patří: PDF, RTF, PNG, PS atd.

Skript používá funkci *system*, která vykoná systémový příkaz a vrátí jeho výsledek. Aby vše fungovalo, je třeba aby server povoloval volání této funkce, ta může být z bezpečnostních důvodů zakázána. Skript funguje tak, že nejprve na serveru v adresáři *Temp* vytvoří soubor, do kterého zapíše zdrojová data. Do tohoto adresáře je také uložen ztransformovaný dokument a pomocí JS je pak momentální URL adresa přesměrována na adresu výsledného dokumentu. Prohlížeč tak otevře výsledný

dokument a v případě, že to nedokáže (např. u formátu PostScript), otevře dialog pro přenos souboru.

4.4 Testování editoru

Chování editoru ve WYSIWYG režimu částečně závisí na použitém prohlížeči. Editor jsem testoval v prohlížečích *Microsoft Internet Explorer 6.0* (IE), *Mozilla FireFox* (FF) verze 5.0 a *Opera* verze 9.26.

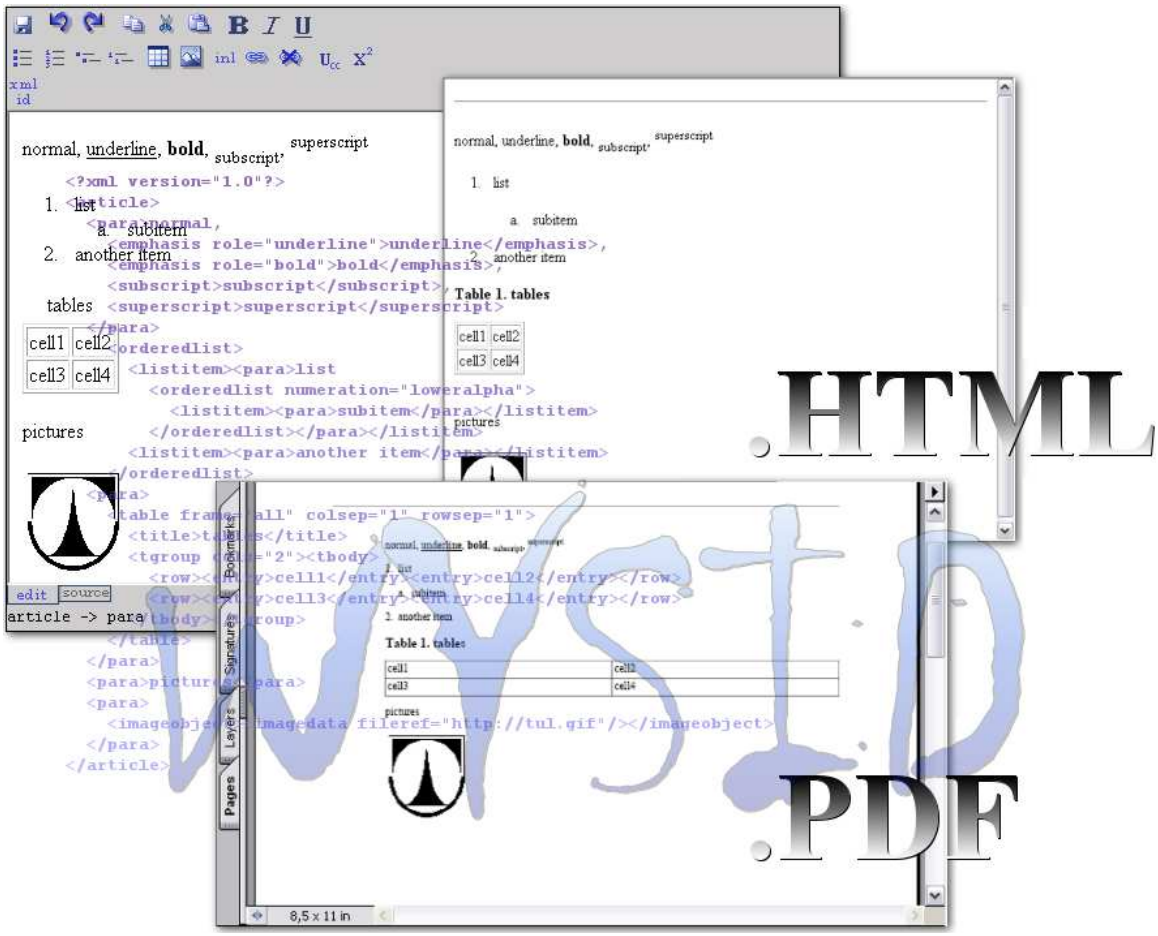
U DB jako u XML jazyka je narozdíl od HTML důležité členit text do odstavců. Opera však reaguje na stisk klávesy *enter* tak, že do dokumentu přidá HTML element *br*, namísto aby vložila nový odstavec jako je to u IE a FF. Co se týče členění odstavců nejlépe se chová IE. Na druhou stranu v IE nefunguje spolehlivě funkce *zpět* a *vpřed*. Prohlížeče se také liší podporou příkazů, které nabízí jejich rozhraní pro WYSIWYG editory. Navíc HTML element *textarea* použitý pro realizaci non-WYSIWYG režimu editoru vykazuje v prohlížeči FF chyby. Ty spočívají v tom, že při opakovaném zápisu textového řeťezce do obsahu *textarea* pomocí skriptu nezobrazí FF nový obsah. Tato chyba platí i opačně, při změně textu uživatelem je ve vlastnosti *textContent* objektu *textarea* stále starý text.

Další otázkou je implementace elementů *iFrame* a *Object* v jednotlivých prohlížečích. Element *Object* obstál poměrně dobře jako náhrada za element *iFrame* ,problémy nastávají při jeho dynamickém generování skriptem. Editor se chová stejně při použití obou elementů, rozdíl je pouze v určitých referencích (instance objektů *document* a *window*), které používá skript pro získání důležitých dat v paměti počítače. Uživatel implementující editor na své stránky si může vybrat, kterým z těchto elementů se bude editor generovat.

5 Závěr

Vzhledem k současné situaci na poli podpory standardů prohlížeči webových stránek je vytvoření WYSIWYG editoru běžícího ve webové stránce pro „*neHTML*“ značkovací jazyk bohužel takřka nemožné. Pokud bychom chtěli napsat takový editor, je možno ho programovat jako Java aplet (softwarová komponenta běžící v kontextu jiného programu). V případě, že by editor musel být napsán v JS, by se takový editor dal poměrně dobře napsat jako non-WYSIWYG editor zobrazující zdrojový kód, případně i rozbalovací strom dokumentu s možností rychlého náhledu na dokument.

Z těchto důvodů jsem při psaní editoru použil relativně netradiční postup. Ten se opírá o rozhraní W3C DOM, ale nevyžaduje ke své činnosti podporu XSLT v prohlížeči. Díky faktu, že se JS příliž nepoužívá ve spojení s problematikou WYSIWYG editorů pro XML, nabývá projekt určité jedinečnosti.



Obr. 5.1: Editor v akci

Editor lze stáhnout ze stránek sourceforge.net.

Literatura

[1] MEYER, Eric A. *DocBook XSL: The Complete Guide*. Santa Cruz: Sagehill Enterprises, 2007. ISBN 0-97415-213-7

[2] WALSH, Norman – MUELLNER, Leonard. *DocBook: The Definitive Guide*. Cambridge: O'Reilly Media, 1999. ISBN 1-56592-580-7.

[3] Docbook.cz [online]. URL: <http://docbook.cz>.

[4] JavaScript tutorial [online]. URL: <http://www.howtocreate.co.uk/tutorials/javascript/>.

[5] Jiří Kosek [online]. URL: <http://www.kosek.cz>.

[6] PHP: Hypertext Preprocesor [online]. URL: <http://www.php.net>.

[7] W3Schools XML tutorial [online]. URL: <http://www.w3schools.com/xml/>.

[8] World Wide Web Consorcium [online]. URL: <http://www.w3.org>.

Příloha

Vybrané objekty

Objekt

- Dceřiný objekt
- Dceřiný objekt tvořící součást pole
- Vlastnost objektu
- Metoda objektu

komentář

Legenda

Javascript

- Array pole
  - length velikost pole
  - push(novy_element) přidá do pole *novy_element*
  - sort() seřadí vzestupně prvky pole
- String textový řetězec
  - length délka řetězce
  - indexOf(hledany_retezec) vráti pozici hledaného řetězce
  - slice(odsazeni_zleva,odsazeno_zprava) vrátí podřetězec podle odsazení
  - split(oddelovac) vrátí pole tvořené prvky oddělené znakem *oddelovac*
  - toLowerCase() vrátí řetězec s malými písmeny místo velkých

W3C DOM

Vytváření větví

document

- createElement(jmeno_elementu) vytvoří instanci větve typu element
- createTextNode(text) vytvoří instanci větve typu text

Procházení stromem

document

- body větev *body* u HTML dokumentu
- documentElement *root* element dokumentu
- getElementById(id) vrátí element na základě jeho *id*
- getElementsByTagName(tag) vrátí pole elementů podle jména tagu

node větev

childNodes[] pole dceřiných větví

parentNode otcovská větev

firstChild to samé jako *childNodes[0]*

nextSibling následující sousední větev

previousSibling předchozí sousední větev

Připojování, mazání větví

node

appendChild(node_ref) připojí větev *node_ref* k větvi

insertBefore(nodeX,nodeY) připojí větev *nodeX* před větev *nodeY*

removeChild(node_ref) odstraní dceřinou větev *node_ref*

Informace o větvi

node

id *id* větve (pokud jde o element)

nodeType druh větve

nodeValue hodnota větve

tagName jméno tagu větve (pokud jde o element)

Atributy

node

attributes[] pole dceřiných větví typu atribut

getAttribute(jmeno_atributu) vrátí hodnotu atributu daného jména

setAttribute(jmeno,hodnota) nastavý atribut daného jména na hodnotu

Můj editor

wysid

docbook

retSource() vrátí zdrojový kód formátu DocBook

loadFromFile(soubor) načte data ze souboru

loadFormXML(data) načte data z textového řetězce

Obsah CD

Dokumentace/BP_Růžička_2008.pdf

elektronická verze bakalářského projektu

Program/index.html

HTML dokument s ukázkou editoru

Program/WYSID/

adresář se samotným editorem

Program/WYSID/readme.txt

návod na použití editoru (jak implementovat editor na stránku, využívat důležité metody, atp.)